Chapter THIRTY
Localization

Exam Objectives

Read and set the locale by using the Locale object.
Create and read a Properties file.
Build a resource bundle for each locale and load a resource bundle in an application.

Localization

Localization (abbreviated as l10N because of the number of characters between the first and the last letter) is the mechanism by which an application is adapted to a specific language and region.

It's related to the concept of *internationalization* (abbreviated as i18n for the same reason as localization), which is about designing an application that can handle different languages and regions.

The most common things that can be customized by language and/or region are messages, dates, and numbers.

In Java, it all starts with one class, `java.util.Locale`.

The `Locale` class basically represents a language and a country although, to be precise, a locale can have the following information:

- An ISO 639 alpha-2 or alpha-3 language code, like *ja* (Japanese)
- An ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code, like *JP* (Japan)
- A variant name, usually empty but can be any string.
- An ISO 15924 alpha-4 script code, like *Latn* (Latin)
- A set of extensions represented by single characters, like *u*.

But most of the time, we just work with languages and countries.

A Locale representation

The language part is required

The country part is optional

fr_CA

Notice the lower case in the language part

Then the underscore for separation

Notice the upper case in the country part

You can get the default locale of your machine with:

```
Locale locale = Locale.getDefault();
```

And get information like:

```
System.out.println("Country Code: "
                        + locale.getCountry());
System.out.println("Country Name: "
                        + locale.getDisplayCountry());
System.out.println("Language Code: "
                        + locale.getLanguage());
System.out.println("Language Name: "
                        + locale.getDisplayLanguage());
```

The output (notice how the actual names are localized in Spanish):

```
Country Code: MX
Country Name: México
Language Code: es
Language Name: español
```

You can also get all the locales supported by Java:

```
Locale [] locales = Locale.getAvailableLocales();
Arrays.stream(locales)
         .forEach(System.out::println);
```

This would output around 160 locales in the form of `language[_country]`, for example:

```
it
pt_BR
ro_RO
```

Setting the locale

There are three different ways to create a `Locale` instance:

1. Using a constructor
There are three constructors:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

For example:

```
Locale chinese = new Locale("zh");
Locale CHINA = new Locale("zh", "CN");
```

2. Using the forLanguageTag(String) factory method
This method expects a language code, for example:

```
Locale german = Locale.forLanguageTag("de");
```

3. Using Locale.Builder
You can set the properties you need and build the object at the end, for example:

```
Locale japan = new Locale.Builder()
                .setRegion("JP")
                .setLanguage("jp")
                .build();
```

Passing an invalid argument to any of the above three methods will not throw an exception, it will just create an object with invalid options that will make your program behave incorrectly:

```
Locale badLocale = new Locale("a", "A"); // No error
System.out.println(badLocale); // It prints a_A
```

So it's good to know that `Locale` class also provides predefined constants for some common languages and countries, for example:

```
Locale.GERMAN
Locale.KOREAN
Locale.UK
Locale.ITALY
```

For the exam, you don't have to know all these constants, or some obscure language and country codes, just that there are four ways to start working with locales.

Once you have a `Locale` object, you can change the locale of your program with the `setDefault(Locale)` method:

```
System.out.println(Locale.getDefault()); // Prints let's say en_GB
Locale.setDefault(new Locale("en", "US"));
System.out.println(Locale.getDefault()); // Now prints en_US
```

Property files

Property files define strings in key/value pairs separated by lines.

There are some rules, like:

- Spaces at the beginning of the line (if any) are ignored.
- Any line that starts with `#` or `!` will be treated as a comment.
- You can break a line for readability purposes with a backslash.

For example, we can have the following file:

Messages.properties

```
# Video related messages
video.added = The video has been added
video.deleted = The video has been deleted
```

To read it, you create an instance of `java.util.Properties` and load it with either a `java.io.Reader` or a `java.io.InputStream`, for example:

```
Properties prop = new Properties();
try (InputStream is = getClassLoader()
            .getResourceAsStream("Messages.properties")) {
    prop.load(is); // Load properties
    // Prints The video has been adde
    System.out.println(prop.getProperty("video.added"));
    // It prints a default value if the key is not foun
    System.out.println(prop.getProperty("video.add", "default"));
    // Get all properties keys
    Enumeration<?> e = prop.propertyNames();
} catch (IOException e) {
    e.printStackTrace();
}
```

Resource Bundles

To localize an application, we have Resource Bundles, which define a set of keys with localized values. Resource Bundles can be property files or classes.

To support this, we have an abstract class `java.util.ResourceBundle` with two subclasses:

java.util.PropertyResourceBundle
Each locale is represented by a property file. Keys and values are of type `String`.

java.util.ListResourceBundle
Each locale is represented by a subclass of this class that overrides the method `Object[][] getContents()`. The returned array represents the keys and values. Keys must be of type `String`, but values can be any object.

In both methods, the name (of the file or the class) follows a convention which allows Java to search for resource bundles and match them to their corresponding locales.

That name convention is:

package.Bundle_language_country_variant

For example:

com.example.MyBundle_fr_FR

Only the resource bundle name is required (and the name of the package if it is not the default one).

For example, we can have bundles with the following names (assuming we're working with property files, although it's the same with classes):

```
MyBundle.properties
MyBundle_en.properties
MyBundle_en_NZ.properties
MyBundle_en_US.properties
```

To determine which bundle belongs to a particular locale, Java tries to find the most specific bundle that matches the properties of the locale.

This means that:

1. Java first searches for a bundle whose name matches the complete locale:
`package.bundle_language_country_variant`

2. If it cannot find one, it drops the last component of the name and repeats the search:
`package.bundle_language_country`

3. If it cannot find one, again, it drops the last component of the name and repeats the search:
`package.bundle_language`

4. If still cannot find one, the last component is dropped again, leaving just the name of the bundle:
`package.bundle`

If nothing is found, a `MissingBundleException` is thrown.

If a class and a property file share the same name, Java gives priority to the class.

But there's another important point.

In your program, you can use the keys of the matching resource bundle and **ANY** of its **PARENTS**.

The parents of a resource bundle are the ones with the same name but fewer components. For example, the parents of `MyBundle_es_ES` are:

```
MyBundle_es
MyBundle
```

For example, let's assume the default locale `en_US`, and that your program is using those and other property files, all in the default package, with the values:

MyBundle_EN.properties
```
s = buddy
```

MyBundle_es_ES.properties
```
s = tío
```

MyBundle_es.properties
```
s = amigo
```

MyBundle.properties
```
hi = Hola
```

We can create a resource bundle like this:

```
public class Test {
    public static void main(String[] args) {
        Locale spain = new Locale("es", "ES");
        Locale spanish = new Locale("es");
        ResourceBundle rb = ResourceBundle.getBundle("MyBundle", spain);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));
```

```
        rb = ResourceBundle.getBundle("MyBundle", spanish);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));
    }
}
```

The output:

```
Hola tío
Hola amigo
```

As you can see, each locale picks different values for key `s`, but they both use the same for `hi` since this key is defined in their parent.

If you don't specify a locale, the `ResourceBundle` class will use the default locale of your system:

```
ResourceBundle rb = ResourceBundle.getBundle("MyBundle");
System.out.format("%s %s\n",
    rb.getString("hi"), rb.getString("s"));
```

Since we assume that the default locale is `en_US`, the output is:

```
Hola buddy
```

We can also get all the keys in a resource bundle with the method `keySet()`:

```
ResourceBundle rb =
    ResourceBundle.getBundle("MyBundle", spain);
Set<String> keys = rb.keySet();
keys.stream()
    .forEach(key ->
        System.out.format("%s %s\n", key, rb.getString(key)));
```

The output (notice it also prints the parent key):

```
hi Hola
s tío
```

If instead of using property files we were using classes, the program would look like this:

```
package bundles;
public class MyBundle_EN extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "buddy" }
        };
    }
}

package bundles;
public class MyBundle_es_ES extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "tío" }
        };
    }
}
```

```
package bundles;
public class MyBundle_es extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "amigo" }
        };
    }
}

package bundles;
public class MyBundle extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "hi", "Hola" }
        };
    }
}

public class Test {
    public static void main(String[] args) {
        Locale spain = new Locale("es", "ES");
        Locale spanish = new Locale("es");

        ResourceBundle rb =
            ResourceBundle.getBundle("bundles.MyBundle", spain);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));

        rb = ResourceBundle.getBundle("bundles.MyBundle", spanish);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));
    }
}
```

The only thing that changed in the `Test` class was the name of the bundle (we had to reference the package). This should not surprise you, after all, both `PropertyResourceBundle` and `ListResourceBundle` inherit from the same class.

Remember also, when using classes we can have values of types other than `String`, for example:

```
public class MyBundle extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
                { "hi", "Hola" },
                { "number", new Integer(100) }
        };
    }
}
```

To get an object value, we use:

```
Integer num = (Integer)rb.getObject("number");
```

Instead of `rb.getString(key)`. In fact, this method is just a shortcut to:

```
String val = (String)rb.getObject("hi");
```

Key Points

- *Localization* (abbreviated as I10N because of the number of characters between the first and the last letter) is the mechanism by which an application is adapted to a particular language and region.
- The `java.util.Locale` class basically represents a language and a country, and is the starting point for localization in Java.
- You can get the default locale of your machine with:

```
Locale locale = Locale.getDefault();
```

- You can also get all the locales supported by Java:

```
Locale [] locales = Locale.getAvailableLocales();
```

- You can create a `Locale` object using a constructor:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

- By using the `forLanguageTag(String)` factory method:

```
Locale german = Locale.forLanguageTag("de");
```

- By using `Locale.Builder` :

```
Locale japan = new Locale.Builder() .setRegion("JP") .setLanguage("jp") .build();
```

- By using predefined constants for some common languages and countries, for example: `Locale.GERMAN`
  `Locale.KOREAN`
- Once you have a `Locale` object, you can change the locale of your program with the `setDefault(Locale)` method:

```
Locale.setDefault(new Locale("en", "US"));
```

- Property files define strings in key/value pairs separated by lines.
- To localize an application, we have Resource Bundles, which define a set of keys with localized values. Resource Bundles can be property files or classes.
- **java.util.PropertyResourceBundle**. Each locale is represented by a property file. Keys and values are of type String.
- **java.util.ListResourceBundle**. Each locale is represented by a subclass of it that overrides the method `Object[][] getContents()` . The returned array represents the keys and values. Keys must be of type `String` , but values can be any object.
- To determine which bundle belongs to a particular locale, Java tries to find the most specific bundle that matches the properties of the locale.
- If it cannot locate one, the last component of the name is dropped until it's just the name of the bundle.
- If nothing is found, a `MissingBundleException` is thrown.
- If a class and a property file share the same name, Java gives priority to the class.
- You can use the keys of the matching resource bundle and **ANY** of its **PARENTS**. The parents of a resource bundle are the ones with the same name but fewer components.

Self Test

1. Given:

```
public class Question_30_1 {
    public static void main(String[] args) {
        Locale locale = new Locale("", "");
        ResourceBundle rb = ResourceBundle.getBundle("Bundle1", locale);
        System.out.println(rb.getString("key1"));
```

```
    }
}
```

Bundle1.properties

```
key1 = Hi
```

What is the result?
A. `Hi`
B. `null`
C. Compilation fails
D. An exception occurs at runtime

2. Which of the following are valid ways to create a locale?
A. `new Locale();`
B. `Locale.Builder().setLanguage("de");`
C. `new Locale.Builder().setRegion("DE").build();`
D. `Locale.forRegionTag("it");`

3. Assuming a default locale `de_DE` , which of the following resource bundles will be loaded first with

```
ResourceBundle rb = new ResourceBundle("MyBundle");
```

A. `MyBundle.class`
B. `MyBundle.properties`
C. `MyBundle_de.class`
D. `MyBundle_de.properties`

4. Given:

```
public class Question_30_4 {
    public static void main(String[] args) {
        Locale locale = new Locale("en", "CA");
        System.out.println(rb.getString(locale));
    }
}
```

What is the result?
A. `en`
B. `en_CA`
C. `CA`
D. `CA_en`

5. Which of the following are valid ways to get a value given its key from a property file resource bundle `rb` ?
A. `rb.getValue("key");`
B. `rb.getProperty("key");`
C. `rb.getObject("key");`
D. `rb.get("key");`

Open answers page

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

29. JDBC API A1. From Java 6 to Java 8